WEST

Help | Logout | Interrupt

Main Menu | Search Form | Posting Counts | Show S Numbers | Edit S Numbers | Preferences | Cases

Search Results -

Terms	Documents
(false adj clause$ or false adj statement$ or false adj predicate$)	3

US Patents Full-Text Database
US Pre-Grant Publication Full-Text Database
JPO Abstracts Database
EPO Abstracts Database
Derwent World Patents Index

Database: IBM Technical Disclosure Bulletins

Search: L1

Refine Search

Recall Text | Clear

Search History

DATE: Tuesday, October 07, 2003 Printable Copy Create Case

Set Name side by side	**Query**	**Hit Count**	**Set Name** result set
	DB=TDBD; PLUR=YES; OP=OR		
L1	(false adj clause$ or false adj statement$ or false adj predicate$)	3	L1

END OF SEARCH HISTORY

WEST

Generate Collection | Print

Search Results - Record(s) 1 through 3 of 3 returned.

☐ 1. Document ID: NNRD438140

L1: Entry 1 of 3 File: TDBD Oct 1, 2000

TDB-ACC-NO: NNRD438140

DISCLOSURE TITLE: Lightweight Run-Time Conditional Switches on IA64

PUBLICATION-DATA:
IBM technical Disclosure Bulletin, October 2000, UK

ISSUE NUMBER: 438
PAGE NUMBER: 1849

PUBLICATION-DATE: October 1, 2000 (20001001)

CROSS REFERENCE: 0374-4353-0-438-1849

DISCLOSURE TEXT:

Disclosed is a method for managing conditional paths in performance critical code on the IA64* architecture. The disclosure specifically addresses the need in an operating system kernel to have highly optimized code paths yet allow for flexibility as key features of the operating system are enabled or disabled. The method employs existing technologies provided in the IA64 architecture including special purpose kernel registers and predication. In software in general, especially operating systems and performance critical code, it is desirable to streamline code paths as much as possible by avoiding compare and branch sequences in performance critical code. In an operating system, especially one where a single binary image of the operating system supports all possible platforms, modes, and settings, it becomes difficult to streamline this code and at the same time conditionally support all of the various features. The invention is specific to software running on the IA64 architecture, and is described in the context of how it was implemented which was for kernel level conditional switches. A kernel flag word is maintained that contains a bit for any run-time decision that might need to be made in performance critical (state management, system call, etc.) code. An example of these bits are for Performance Tracing Enable/Disable, System Trace Enable/Disable, Lowlevel Kernel Tracing Enable/Disable, System Call Tracing Enable/Disable, C2 security or better enabled/disabled, etc. Each of these conditionals result in a different path, or additional code being executed in performance critical paths. Instead of having to test each of these conditions independently, which could result in multiple separate compare instructions with associated conditional branches, the performance critical code paths load the current value of kernel flags into the IA64 predicate register. Then in that one instruction, all possible conditional results are set in their assigned predicate. Since the kernel flag word is kept in a special register, there aren't even any memory references associated with determining the results of each condition. The performance critical code then contains predicated instructions that branch out-of-line to accomplish the task. If the flag isn't enabled, the corresponding predicate will be FALSE, and the predicated instruction in the mainline path essentially ignored with no performance impact of introducing compares and branches to the normal path. * Trademark of the Intel Corp.

☐ 2. Document ID: NA9109320

L1: Entry 2 of 3 File: TDBD Sep 1, 1991

TDB-ACC-NO: NA9109320

DISCLOSURE TITLE: Sound and Consistent Handling of Nulls in Data Bases.

PUBLICATION-DATA:
IBM Technical Disclosure Bulletin, September 1991, US

VOLUME NUMBER: 34
ISSUE NUMBER: 4A
PAGE NUMBER: 320 - 330

PUBLICATION-DATE: September 1, 1991 (19910901)

CROSS REFERENCE: 0018-8689-34-4A-320

DISCLOSURE TEXT:

- The concept of a "null" field in a data base is ill-defined, and nulls give rise to a number of problems in relational DBMs. A technique is described that provides a definition and leads to solutions. - It is sometimes desired to record a field in a data base as having no value, or as having an "improper" value (such as "unknown", or "not yet available"), i.e., as being NULL. However, underlying theory does not support such missing or improper values. Ad hoc adjustments to the theory, or to DBMSs, to allow missing or improper values, threaten the soundness and consistency of the user interface. For example, the current SQL NULL both is ill-defined and requires the use of a three-valued logic. The user of SQL is consequently unable to grasp the meaning of a NULL field, or, when such fields are present, safely to apply the conventional correct truth-valued logic. The technique disclosed here comprises: * A definition of NULL, which is a clear concept and allows a test to be formulated whereby it may be determined of a putative value whether it is NULL or not. * Extensions of the definitions of all relational operations, which permit the uniform application of truth-valued logic. * A means of expressing and handling different kinds of NULLs. SUMMARY OF THE TECHNIQUE With respect to relational theory: * A distinction is drawn between a cell (i.e., a field instance, or column/row intersection in a table), and the value, if any, contained in such a cell.

* The constraint of first normal form (1NF), viz that each cell contains exactly one value, is relaxed to: each cell contains, at most, one value. * All and only cells with no value are said to be NULL. * All operations are defined primarily upon cells, and only secondarily upon the values in those cells. - With respect to the interpretation placed by users on their data: * With each column in a table is associated a criterion of identity, which explicates the meaning of that column, in the sense that, for a column, say C, any rows with the same value in that column can be said to have or be the same C. (For example, two rows representing Persons and having the same value in the column Salary would be said to have the same Salary.) * A putative value in a row and column is said to be "proper" (and is accordingly represented by a value in a cell) if and only if it meets the criterion of identity of the column; otherwise, it is said to be "improper" and represented by an empty cell (a cell with no value, a NULL cell). * Where it is necessary to distinguish between different improper values, or to distinguish different ways (termed modalities) in which a cell (or its value) in some column, say C, pertains to the meaning of the column, it is permitted to introduce an associated column, say C', such that the cell of C' in any row has as value the modality of the cell of C in that row. * In order for operations on cells (or the values in cells) either to vary in their results according to the modalities of those cells or to generate the

modalities of resultant cells, it is permitted to define a partial ordering of the modalities. - In the following description SQL data sublanguage is used by way of illustration. The technique is applicable to other relational languages, and in large part both to non-relational and to extended relational languages, if they have similar underlying theories. RELATIONAL THEORY WITHOUT NULLS Relational Theory is founded on the predicate calculus. First, ignoring NULLs, a table (relation) represents an open sentence, or predicate, that contains one or more places, which may be represented as gaps, as pronouns, or as pronomial phrases that denote either domains or roles, for example: ___ is father of ... - He is father of him/her . - This PERSON is father of that PERSON . - This FATHER is father of that CHILD . - Rows in such a table are obtained by inserting names as values in the places: Isaac is father of Jacob . - Jacob is father of Reuben . - Jacob is father of Dinah . - Thus each row represents a closed sentence or proposition. While a proposition is either true or false, a predicate is true or false of things taken (in this case) pairwise. According to the number of places in the predicate, it may be true of things taken singly, pairwise, triplewise, or n-tuplewise. Usually a table is represented by a name (indicating the predicate), such as FATHERHOOD; column names (indicating the places), such as FATHER, CHILD; and the rows of values inserted: FATHERHOOD FATHER CHILD Isaac Jacob Jacob Reuben Jacob Dinah A relational data base (a collection of tables) therefore comprises a theory about reality (viz, in this case, the theory that Isaac is father of Jacob, and Jacob is father of both Reuben and Dinah). By representing predicates "P(...)", "Q(...)", "R(...)" and so on, and (possibly empty) sets of column values by "a", "b", etc., then a row in a table may be represented as, say, "P(a)" or "Q(a,b)" or "R(a,b,c)". This enables a definition of the relational operations. The first class of these has the general form of the conjunction: R(a,b,c) = P(a,b) AND Q(b,c) where P and R are the predicates represented by tables, and Q is either another such predicate or some other sort of predicate (such as a Restriction condition, as in an SQL WHERE or HAVING clause). 1. Cartesian Product, with no common columns (i.e., b empty) - as defined in an SQL FROM clause R(a,c) = P(a) AND Q(c) 2. Restriction, where Q is the Restriction (WHERE or HAVING) condition, and all the operands of Q are columns of P R(a,b) = P(a,b) AND Q(b) 3. Extension (appending calculated columns in the SELECT clause), where Q(b,c) is a function, viz c=q(b) R(a,b,c) = P(a,b) AND Q(b,c) 4. Intersection (achievable by Cartesian Product and equality Restrictions on all columns, or by the IN operator), with all columns common (a and c empty) R(b) = P(b) AND Q(b) The second class of operations uses a variety of other logical operators: 5. Difference, like Intersection, but AND NOT instead of AND (using NOT IN) R(b) = P(b) AND NOT Q(b) 6. Union, like Intersection, but OR instead of AND (using UNION) R(b) = P(b) OR Q(b) 7. Projection (defined in the SELECT clause) P(a,b) = THERE EXISTS AN x SUCH THAT R(a,b,x) Natural Join, which is represented by the unrestricted conjunction: R(a,b,c) = P(a,b) AND Q(b,c) is not directly achievable in SQL (which does not recognize common columns). It is done by Cartesian Product, Restriction, and Projection: R(a,b,c) = THERE EXISTS AN x SUCH THAT b=x AND P(a,b) AND Q(x,c). That is, SELECT a, P.b, c FROM P,Q WHERE P.b = Q.b. - It can be seen how, given the interpretations of the base (stored) tables, the interpretations of derived tables (those obtained by applying relational operations) are fixed: they are determined by the truth functional operators (AND, OR, NOT) and existential quantification (THERE EXISTS ....). The importance of using names as values should be appreciated. When a name, such as "Jacob", is given (say to a baby), the name is given to anything (anyone) who is the same Person as that baby. It is generally convenient, for the more restricted purposes of defining a table, to use the column name (the role name, rather than the entity name) for this criterion of identity. Thus, say that two persons named as CHILD in different rows of the table are said to have the same Father if the same name is the value in the FATHER column of those rows (as is "Jacob" in the "Reuben" and "Dinah" rows). If something other than a name is used as a value (i.e., something that does not meet the criterion of identity), even though the resultant proposition is grammatically correct, the relational operators no longer give rise to intuitively correct interpretations. Thus, let "someone" be substituted for the FATHER values in the first two rows: FATHERHOOD FATHER CHILD someone Jacob someone Reuben Jacob Dinah These are still true propositions that someone is father of Jacob and that someone is father of Reuben. If this table is joined to itself on the FATHER column, and then project the two CHILD columns, it results in a table representing the predicate: ___ has the same father as .... Having allowed the (improper) value, "someone", such a table would include the row: Jacob Reuben And thus the falsehood that Jacob has the same father as Reuben could be concluded. Use of "someone" is, in effect, the "Unknown" NULL. Other NULLs include the "Not Applicable" NULL - roughly equivalent to "nothing" or "nobody". Again, "nothing" is not a name. Two persons neither of whom has a second given name do not have the same second given name, nor do two persons whose Religion is marked

as "none" have the same Religion. RELATIONAL THEORY WITH NULLS In order to admit NULLs, the relational theory must be extended to distinguish between row/column intersections, termed cells, and the values held in those cells. Formally, treat each cell as a non-compound set: either a unit (one-membered) set or the empty set, thus: FATHERHOOD FATHER CHILD {} {Jacob} {} {Reuben} {} {Dinah} ALL NULLs are represented by the empty set without distinguishing between types of NULL. Now reconsider the relational operators. Recall that expressions like "R(a,b,c)" represented rows in a table, where "a", "b", and "c" represented sets of column values. Now let "a", "b", and "c" represent sets of cells. First consider Restrictions, defined above as: R(a,b) = P(a,b) AND Q(b), where Q is not a table but a predicate. Take a special case where "b" represents two cells, say, b1 and b2, and Q(b) is: b1 = b2. - Because of the new distinction between cells and values, this predicate, "b1 = b2", is now ambiguous. Under one interpretation (which we may term "cell equality") it means: b1 is the same set as b2. - This makes NULL calls to be treated like any others. But under another interpretation (which we may term "value equality" it means: b1 and b2 have (contain) values AND b1 is the same set as b2. - Under value equality, a NULL cell is not equal to any cell (even to another NULL cell). So it is necessary to distinguish between cell and value-operators. For example, in a language like SQL, use current operators, such as "=", ">", etc., as value-operators and "==", ">>", etc., for cell-operators. Notice that in general a value operator, say, "&", is definable (as above), thus: b1 & b2 amounts to: b1 and b2 contain values AND b1 && b2. - If, as is common, an operator exists that is the negation of "Q" (as "NOT =" is of "="): b1 -& b2 amounts to: (b1 and b2 contain values) AND (b1 -&& b2). - So operator negation (predicate negation) is not equivalent to propositional negation: NOT (b1 & b2) amounts to: NOT ((b1 and b2 contain values) AND (b1 && b2)). - That is, (NOT b1 and b2 contain values) OR (b1 -&& b2). - In sum, the effect of NULL cells on Restrictions is the need to introduce two series of operators (cell and value), and to distinguish predicate and propositional negation. Alternatively, make do with value-operators and an "IS (NOT) NULL" cell-operator, defining: b1 && b2 as (b1 IS NULL AND b2 IS NULL) OR b1 & b2. - With query languages, like SQL, operators like "+" and "-" are found as well as predicate operators like "=" and ">" are found. Thus Q(b) might be condition like: b1 + b2 < 100. - Therefore, it must be possible to add NULLs, substract them, and so on. Indeed, the arithmetic operators need redefinition to apply to cells. The redefinition for non-empty cells is easy: the sum of two non-empty cells is defined as a set that contains the sum of their values. The redefinition for all empty cells is equally easy: the sum of two empty cells is defined as the empty set. There is no obvious definition of the result of applying an operation to one empty and one non-empty cell. Each operation needs to have its interaction with empty cells specially defined. One good guideline is that information loss should be minimized. Thus adding or subtracting an empty cell should be equivalent to adding or subtracting the number zero; multiplying or dividing with an empty cell should be equivalent to multiplying or dividing with the number one. In general, an empty cell should work like the identity element of the operation. - It might be desirable to introduce an "IF ... THEN ... ELSE" operator to give the user flexibility (e.g., to make a sum NULL if "+" were applied to a NULL field) thus: (IF b2 IS NULL THEN NULL ELSE b1 + b2) < 100.

- Turning to Projection, defined as: P(a,b) = THERE EXISTS AN x SUCH THAT R(a,b,{x}). - Here, is a similar distinction: what does "x" denote? If we take "x" to denote a cell, then our Projection operation is unchanged. If we take "x" to denote a value, then we must redefine: P(a,b) = THERE EXISTS AN x SUCH THAT R(a,b,{x}. - It is probably unnecessary to introduce this sort of value-projection, and is certainly counter-intuitive in SQL where it is the non- quantified columns that are named: SELECT a, b FROM R Instead, the user could simply apply a Restriction to R: P(a,b) = THERE EXISTS AN x SUCH THAT R(a,b,x) AND x IS NOT NULL That is, SELECT a,b FROM R WHERE c IS NOT NULL. - The Cartesian Product needs no redefinition; also, Intersection, Difference, and Union are probably best left unchanged: all these operations then become cell, rather than value-based. Extension needs no redefinition, although in some cases a NULL cell might be concatenated (when q(b) results in an empty cell). - Joins (other than Natural Join) are automatically redefined, when value-operations are used, because of the redefinition of Restriction. If the definition of Natural Join is unchanged (as it probably should be), the equivalence between Natural Join and a Projection of an Equijoin (i.e., a Join whose Restriction is an equality) breaks down if a value-equality, rather than a cell-equality, is used. As Natural Joins commonly involve key fields which must not be NULL, this is of little importance. In SQL, which has no Natural Join operator, it is of no importance. The user would code one of the following, as required: SELECT a, P.b, c FROM P,Q WHERE P.b = Q.b SELECT a, P.b, c FROM P,Q WHERE

P.b == Q.b The adjustment of relational operations to allow the use of empty cells amounts to little more than a careful redefinition of the operators used in Restrictions. One needs to provide an interpretation of rows that contain empty cells, and consequently of the new rows obtained by applying the operations to them. INTERPRETATION OF NULLS Consider again the table: FATHERHOOD FATHER CHILD {} {Jacob} {} {Reuben} {} {Dinah} It is clear one cannot simply interpret this as: ___ is the father of .... - Instead, we must reinterpret as: {___} contains the name (if recorded) of the father of the child (or person) called by the name (if recorded) contained in {...}. - The expression "the name (if recorded)" indicates that an incomplete function is being used. In some cases this function returns a name, in others it returns nothing. So, for those rows with no NULLs, the two predicates are equivalent. Consider now the case of a row in which the FATHER column is NULL. This amounts to: {} contains the name (if recorded) of the father of the child (or person) called by the name (if recorded) contained in {...}, which is simply: Some child (or person) is called by the name (if recorded) contained in {...}. - Thus, although formally each row in the table is a proposition of the same form (i.e., built from the same predicate) to the user, differ ent rows convey different predicates. However, when a projection is applied that takes only non-NULL columns, a uniform predicate is obtained. Let us suppose that the CHILD column is non-NULL. The common predicate - as understood by the user is "... is a person". - Conjoining this predicate with the original predicate results in: "... is a person AND ___ is (recorded as) father of ..." Conjoining it with the appropriate predicate when the FATHER column is NULL produces: "... is a person AND no father of ... is recorded". - The user must be aware that either of these predicates might be used in a row of the table. But this awareness is nothing but the knowledge that the FATHER column may be NULL. The interpretation of tables formed by applying most of the relational operators is then straightforward. The principal change is in the interpretation of Projection. As each (possibly) NULL column has been represented by a separate conjunct, a Projection that removes a NULL column is no longer (from the user view) an existential quantification. Thus, a projection over the CHILD column amounts to: ... is a person AND EITHER no father of ... is recorded OR someone is recorded as father of ... - But this is no more than: ... is a person. - Note that the reinterpretations thus far given depend on having a field (the CHILD) that is not NULL. And, in stored tables, this is guaranteed by relational theory (the key must not be NULL). Even if Projection is used to remove such fields, an interpretation can still be applied to the resultant relation because a Projection that removes a key can always be interpreted as an existential quantification. So, projecting over the FATHER column (and removing the key CHILD column) gives rows formed from one of the sentences: Someone is a person AND ___ is (recorded as) father of that person. - Someone is a person AND no father of that person is recorded. - There can, of course, be no more than one row formed from the latter (the all NULL row): it is already a closed sentence. PROPRIETY OF VALUES The principal problems of NULLs arise from their use in Joins (most of which are equijoins - Joins with an equality Restriction), and in keys, where they are so troublesome as to be forbidden. These are two aspects of the same problem. Data base design is in part the separation of tables that can, without information loss, be held separately and joined on their keys. Also shown above is how improper values fail to meet the criteria of identity of their columns, and it will be appreciated why it is critical that keys do not contain such values: they are specified precisely as identifiers. According to the technique, a NULL value is an improper value: one that fails to meet the criterion of identity of its column. If a value is (in this sense) improper, then, irrespective of its interpretation, it gives rise to the problems characteristic of NULLs. It should also be appreciated that if a value, no matter how "NULL-like" its interpretation, does meet the criterion of identity of its column, then it will not give rise to these problems, and should not be treated as NULL. It is not only ineffective, but misleading, to identify NULL with any or all of a list of interpretations, like "None", "Unknown", "Not Yet Applicable", and so forth. The rule of column criterion of identity is, in contrast, not only a clear definition of nullity, but also an effective test, assuming that the data analyst's task is complete or, in some cases, showing most usefully that there remains analysis to be completed. - Thus far a formal treatment of NULLs has been defined, and a test for NULLs given. However, only one formal representation of NULLs - the empty set - has been defined, while there are (in theory) indefinitely many kinds of NULL. According to the technique, the representation of different kinds of NULL is properly considered as a problem entirely orthogonal to that of representing NULL (missing) values. It is true that, if a column (say, Second Given Name) is NULL, there may be several different reasons for, or comments on, that column being NULL. For example, it might be that the person had no second given name, that they refused to reveal it, that it was known they had a second given name but not what that name was, that

nothing was known about it at all. But, even if a second given name were recorded, we might still wish to make some comment on it. For example, it might be that it was conjectural, possibly incorrectly spelled, not really a given name but a patronymic, possibly merely a nickname. The substance of any comments made about how the cell and its value, if any, attaches to the row (or the thing represented by the row) may be called its modality. MODALITY The modality of a cell may be recorded as a (proper) value in a cell in an associated column. Such an associated column might have its own modality (to be recorded in yet another column), but this is unlikely in practice, and definition of yet one more column could probably be avoided by some compromise in design. Given a column, C, let its associated column be C'. In practice, the association would be made by means of a naming convention, or by treating C' as a function of C, e.g., referring to C' as, say, M(C). Notice that if the value of C' (the modality of C) were calculable from the cell of C, it would not be necessary to store the column C', where, for instance, all non-empty cells of C had the same modality and all empty cells of C had the same modality. Recall that certain operations have to be redefined to allow them to apply to cells rather than values, and specifically to NULL cells. By introducing modalities, operations on, say, columns C and D, generating column E, may also need to generate the new modality column, E', and - to this end - may need to refer to the old modality columns, C' and D'. In general, the application of any operation loses or does not gain information. Thus, if an operation, say @, results in the assignment: E := C @ D, that E has the modality E' will be implied by C having the modality C' (or D having the modality D'), but not vice versa. For example, if: GIVEN-NAME := FIRST-GIVEN-NAME SECOND-GIVEN-NAME (where " " means concatenation with blank in between), then if the modality of FIRST-GIVEN-NAME is possibly misspelled, and that of SECOND-GIVEN-NAME is conjectural, then the modality of GIVEN-NAME is possibly misspelled and/or conjectural. - For each pair of modalities, say, m and n, there must be defined a modality, m n, which results from applying any operation to cells of those modalities. Notice that m n may itself be m or n, and that there will generally be a default (best) modality, say, b, such that for any modality, m, m b (or b m) is m. All modalities will therefore form a partial ordering, say, <=, where m<=n means that m is, roughly, at least as doubtful as (no more reliable than) n. In addition, there is a constraint on this ordering which ensures, given any m and n, a unique value for m n: For any m and n, there is a modality, x (viz m n), such that x

Full | Title | Citation | Front | Review | Classification | Date | Reference | Sequences | Attachments | KWIC
Draw Desc

☐ 3. Document ID: NA89116

L1: Entry 3 of 3 File: TDBD Nov 1, 1989

TDB-ACC-NO: NA89116

DISCLOSURE TITLE: Evaluation of Quantified Predicates With Non-Correlated Subqueries

PUBLICATION-DATA:
IBM Technical Disclosure Bulletin, November 1989, US

VOLUME NUMBER: 32
ISSUE NUMBER: 6A
PAGE NUMBER: 6 - 9

PUBLICATION-DATE: November 1, 1989 (19891101)

CROSS REFERENCE: 0018-8689-32-6A-6

DISCLOSURE TEXT:

- This invention provides a method to evaluate certain predicates that involve subqueries without having to access the subquery or a temporary table containing the subquery. Background In a relational database management system, a predicate (a condition that specifies which rows of a table are to be retrieved) can contain a subquery. That is, data that is to be selected from a table is used in determining the result of the predicate. For example, a query could request the rows of the "employee" table for which the salary column is greater than every value in the salary column of the managers' rows in the "employee" table: select * from employee where salary > all (select salary from employee where job='Manager') A predicate that uses the keyword ALL or ANY (or SOME which means the same thing as ANY) before a subquery is called a quantified predicate. - A straightforward method of evaluating a quantifed predicate would be to compare the value to the left of the comparison operator with each value from the subquery. If ALL is used in the predicate, the comparisons would continue until a false comparison was found -- then the predicate would be false. If all the comparisons were true, the predicate would be true. If none were false, but one or more were unknown (a comparison involved a null value), the predicate is unknown. If ANY is used, the comparisons would continue until a true comparison was found -- then the predicate would be true. If all the comparisons were false, the predicate would be false. If none were true, but one or more were unknown, the predicate is unknown. - This straightforward method can be very inefficient. It could require many accesses of the subquery for every evaluation of the predicate. The Invention If a quantified predicate uses one of the comparison operators <, <=, >, or >=, and if the subquery is not correlated to the same level of the query as the predicate is used in, it can be evaluated when just the following information about the subquery result is known: 1. whether any non-null values were in the subquery, 2. whether any null values were in the subquery, and 3. if non-null values were found, what the highest or lowest value was. - If the predicate is ALL, or >=ALL, the highest value is needed. If the predicate is >ANY, >=ANY, ANY, the following information about the subquery result is needed: 1. whether any non-null values were in the subquery, 2. whether any null values were in the subquery, 3. if non-null values were found, whether there is more than one unique value, and 4. the value, if only one value was found. - All of this information can be computed when the query is opened and saved in pre-allocated memory. Following is the pseudocode for computing this information: Reset all flags Set the subquery result value (SRV) to null Obtain the first subquery value (SV) Until end of file or done flag If the SV is null Set the contains-nulls flag Else Set the contains-non-nulls flag If the SRV is null Set the SRV to the SV Else Compare the SV to the SRV Switch on Predicate Case =ALL or <>ANY If the SV was not equal to the SRV Set the multi-valued flag Set the done flag Endif

Case >ANY, >=ANY, ALL or >=ALL If the SV was greater than the SRV Set the SRV to the SV Endif Endswitch Endif Endif Obtain the next subquery value (SV) Enduntil Since all information needed to evaluate the predicate is now saved in pre-allocated memory, to evaluate the predicate there is no need to access data on disk, nor is there a need to look at all of the values returned by the subquery. The evaluation is done by examining the information saved about the query, as explained above, and the LHS value (value to the left of the comparison operator). Following is the pseudocode for this evaluation process: If the subquery was empty (no null values and no non-null values) then If the quantifier is ALL then Set the result to TRUE Else Set the result to FALSE Endif Else If (the LHS value is NULL) or (the subquery contains no non-null values) or (the quantifer is ALL and the subquery contains nulls) then Set the result to UNKNOWN Else If the subquery has multiple non-null values and the predicate is =ALL or <>ANY then If the predicate is <>ANY then

Set the result to TRUE Else Set the result to FALSE Endif Else: Compare the LHS value with the subquery result value Switch on Comparison Result Case LHS value < subquery result value If the comparison operator is <, <=, or <> then Set the result to TRUE Else Set the result to FALSE Endif Case LHS value = subquery result value If the comparison operator is <=, >=, or = then Set the result to TRUE Else Set the result to FALSE

Endif Case LHS value > subquery result value If the comparison operator is >, >=, or <> then Set the result to TRUE Else Set the result to FALSE Endif Endswitch Endif Endif Endif